6

SOFTWARE AGENTS FOR ROLE BASED SECURITY

S. Demurjian, Sr., Y. He, T. C. Ting and M. Saba *

Abstract In the age of information technology, organizations of all types are seeking to effectively utilize and disseminate information via dependable and secure distributed computing environments. While many existing access control approaches (mandatory, discretionary, and role-based) can be leveraged for the support of security, their assumptions of a centralized computing model may be insufficient in a distributed setting. In recent years, agent computing has emerged as a new computing paradigm, particularly suited to distributed and web-based applications. This paper explores the ability of software agents to support role-based security in a dynamic, object-based setting which is suitable for distributed and web-based applications, with experimental prototypes using Aglets, a Java-based mobile agent model from IBM. The agent approaches differ in their utilization of agents (stationary and mobile) and the granularity level of the involved classes/objects.

Keywords: Agents, Aglets, role-based access control

1. INTRODUCTION

The fundamental challenge facing organizations, software architects, system designers, and application builders (i.e., stakeholders) involves the ability to leverage computing and networking resources, and data and information in existing applications, to construct new dependable and secure distributed and web-based applications. Distributed and web-based applications will require researchers and practitioners to design security solutions that expand and transcend available alternatives. Traditional alternatives such as *mandatory access*

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35508-5_22

V. Atluri et al. (eds.), *Research Advances in Database and Information Systems Security*

control (MAC) [7, 8], *discretionary access control (DAC)* [10, 11], and *role-based security (RBS)* [1] may all be useful as stakeholders and security engineers work towards effective security solutions. Agent computing, which first emerged just five years ago [3], has great potential for supporting the design and development of secure distributed and web-based applications.

As computing objects, software agents have a specific function or responsibility to perform, and can be defined in formal terms to have a state and a behavior within the runtime environment. Software agents have four mandatory properties [9]: ability to sense and react to changes in the environment; autonomous control over its own state and behavior; proactive to achieve specific goals (typically of the user); and, constantly executing within the runtime environment. Stationary agents are restricted to a single computing node. Mobile agents can migrate to other locations. All agents are like objects in that they can be created and destroyed. But, agents cannot invoke each others methods; rather, they communicate via message passing.

Our intent is to present agent approaches to security within a role-based context for distributed and web-based applications. We view this as a crucial first step to understand the way that agent computing models can be utilized to support role-based security. Moreover, if agent computing continues to increase in popularity and usage, it is incumbent upon the research community to provide security solutions for this and other emerging technologies. While the technology is dangerous from a security perspective, it is critical that we offer strong and proven solutions to guarantee varying levels of secure access in a distributed setting that utilizes agents.

The work presented in this paper is a progression of our own efforts [1, 2, 12], with the justification to pursue this effort influenced by other researchers [4, 5, 13]. In our previous work [2], we explored static software architectures alternatives for role-based security, which is inadequate for true dynamic, distributed and web-based applications. Another effort to investigate security for distributed objects [4], has recently been extended to provide a secure distributed object and language programming framework for internet-based applications [5]. Another effort has utilized the Distributed Object Kernel (DOK) project as the underlying framework upon which software agents can operate for the design and implementation of distributed security policies [13]. Our most recent effort has examined the capabilities and the potential impact of Java on security for distributed computing [12].

The goal of this paper is to explore the potential of agent computing in support of role-based security in a dynamic, object-based setting. The remainder of this paper is organized into three sections. In Section 2., we detail and compare three architectures of agent approaches for role-based security in a dynamic, object setting. In Section 3., we detail our experimental prototyping

efforts using one of the architectures presented in Section 2. to clearly demonstrate the feasibility of agent-solutions. In Section 5., we conclude by offering insight into ongoing efforts and future plans.

2. AGENT APPROACHES TO ROLE-BASED SECURITY

Security for distributed and web-based applications must occur in an environment that may consist of legacy, COTS, database, and new/existing server applications, with clients interested in obtaining access to the remote objects that reside in these various applications. One interesting aspect of security is that its goals are often orthogonal at some level to the goals of distributed computing and web-based computing. Security has the goal of controlling and limiting certain types of interactions while distributed and web-based computing is concerned with enabling interoperation and facilitating access to information. Stakeholders must carefully balance the sometimes contradictory goals of security and distribution requirements when constructing distributed computing applications.

For discussion purposes within this paper, we are assuming a client/server computing model, where a client application is interested in sending a request to access/modify a remote object, with the allowable actions on the remote object dictated by the role of the user (client). In our role-based model for DAC [1], we have proposed the concept of a potential public interface for all classes, which contains those methods that are most likely to be made public on a role-by-role basis. The basic premise is that, for each role, we track the positive permissions, namely, can a user role invoke a particular method on an object. Our assumption in this paper is that the client will play a role and make a request to invoke a method on a remote object. Agents will be dispatched to attempt to carry out that request, with a success or failure result.

Thus, the scenario is that a client application is making a request (i.e., a method invocation) to access/modify a remote object, that may be residing in a legacy, COTS, database, or server application. In such a scenario, we want the request to be processed according to the defined security policy for the client application in a dynamic setting. In our role-based security model, this requires that, for each user role, we maintain a list of the permissions (which methods can be invoked) on all classes in an application. Agents are attractive in such a setting, since they allow autonomous and mobile actions to occur on behalf of a client. In our case, these actions will authenticate the identity and role of the client application, will bundle the client request (method invocation) and user role within a message, and then dispatch agents that can move across computing nodes for the secure access of remote objects.

In this section, we present three agent approaches that support role-based access for distributed and web-based applications: a *baseline agent approach* of core system components and agents (both stationary and mobile) to facilitate the remote access to an object; a *hierarchical agent approach* that expands the baseline approach to spawn a series of multiple agents for simultaneously accessing remote objects in multiple locations; and, an *object-security manager agent approach* that extends the hierarchical approach to include the access of multiple remote objects in the same location. The remainder of this section reviews each approach in detail, followed by a summary that compares the three approaches.

2.1 BASELINE AGENT APPROACH

The architecture for the baseline agent approach is presented in Figure 11, and is partitioned into a client computing node (upper half of figure) and a

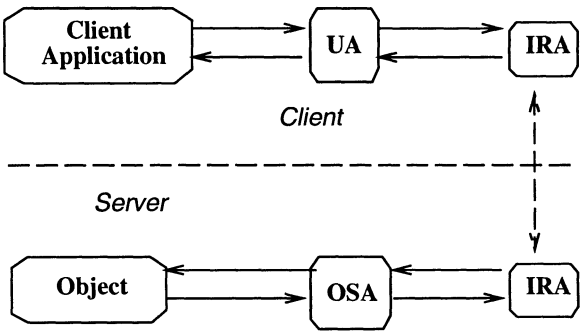

Figure 6.1 Architecture for baseline agent approach.

server computing node (lower half of figure). The components and agents are as follows:

- **Client Application (CA):** This component is the GUI utilized by the user. The user is restricted to playing a single role at any time, and both the role and request are passed from CA to the user agent (UA) for processing. Users can access/modify only a single remote object at a time via CA. If multiple remote objects are to be accessed/modified, there must be functionality within CA that allows such a request to be separated into a series of individual object accesses.

- **User Agent (UA):** This is a stationary agent that is created upon the request of the client to represent the user. UA receives a request from the client, transforms the request to the proper format, creates the information retrieval agent (IRA), forwards the request to IRA, waits for the

processing of the request, collects the results of the request from IRA, and transforms and then returns the results to CA.

- **Information Retrieval Agent (IRA):** This is a mobile agent that is created by UA for processing the request of CA. In this baseline approach, the IRA is limited to interacting with the UA and object security agent (OSA). IRA is created and dispatched by UA to access a single remote object on the server side, if permitted by the role of the user.
- **Object Security Agent (OSA):** This can be a stationary agent (collection of security objects) or a mobile agent. In either case, OSA enforces the security policy of the object that is based on the permissible actions of each role. Each OSA is providing control to a remote object. The *remote object* may be a single object or a collection of objects.
- **Object:** The remote object that is available to provide services to CAs. A remote object may be a single instance or a collection of related instances (e.g., a person database is a single remote object).

The baseline approach is a core strategy with limiting assumptions:

1. The client formulates simple requests that invoke a single method on a single remote object.
2. UA is able to process multiple requests by the client in a synchronous fashion, i.e., there may be multiple simultaneously active IRAs.
3. Authentication of the user (in either CA or by UA) must occur prior to the spawning of an IRA to respond to a request.
4. While a *remote object* refers to a single instance, that instance may be a person record (e.g., name, address, etc.) or a collection instance (e.g., the collection of all person records). In the latter case, multiple instances are controlled like a database within the collection.

The idea for the baseline agent approach is to limit functionality by mandating that complex requests that require synchronization (results of one request drive submittal of successive requests) are handled by the client. Our intent is to provide a simple model for those situations where the majority of the processing already exists in CA, e.g., the client is a legacy/COTS application.

User Agent The *user agent (UA)* is a stationary agent that is utilized when the user requests that an action be initiated on his/her behalf. The user, via CA, can only play a single role at any given time. The UA receives a coded request (i.e., user role and method to invoke) from CA, forwards the request to

IRA, and waits for the response from IRA. For the baseline approach of Figure 11, UA receives a single request to access one remote object. There are two different UA allocations strategies. In *user-based allocation*, a UA is allocated and dedicated to an individual user when the client application commences and the user logs on to a session. Essentially, a dedicated UA can enforce the single role that the client plays and manage all of the synchronous requests that will be submitted to dedicated IRAs (one IRA per request). If multiple CAs are executing on the same client node, then multiple UAs will compete for computing resources. In *role-based allocation*, dedicated UAs are allocated for each role, and shared by the many users (CAs) that are playing the same role. UAs are allocated whenever there is a request for a user (CA) to play a role, and that role is not supported by an active UA. When a UA is no longer being utilized by any active CA, then it can be de-allocated. In both strategies, multiple IRAs may be active and managed by the single UA; in user-based allocation there is one client/UA, while in role-based allocation there is one UA/role.

Information Retrieval Agent The *information retrieval agent (IRA)* is a mobile agent that is created by UA on the client side to process the request of the user (from CA).In the baseline agent approach, the request to be processed by IRA will be limited to a single remote object. As a mobile agent, IRA is able to move to the host computer (server side) where the object resides that is needed to satisfy the user's request. IRA carries the user request (the method to invoke) and the current role, which are both needed to interact with the object-security agent (OSA) to process the request. That is, the OSA will validate whether the IRA is permitted to ask for the method to be invoked on the remote object based on the user role being represented by the IRA. Once the IRA has received a response (success or denied access) from OSA, it takes that response, ceases its action, and moves back to the client side to return the result to UA, which will then forward the response to CA. There are two different scenarios for the lifetime of IRA that share a common basis, IRA is allocated by UA for the first request of a remote object by CA. First, IRA stays alive as long as UA is active. In this case, a single IRA can process all requests by a client in a sequential manner, starting a new request only after the current request has been completed. This scenario reduces the agent management overhead. Second, IRA is de-allocated when it finishes processing a request. In situations were access to remote objects by clients is infrequent, this scenario benefits by not having active, idle agents. The first scenario will not support multiple active requests being spawned by UA. The second scenario supports such a situation, since multiple independent IRAs can be spawned that move to remote locations to access objects. This will require UA to have additional logic to be able to process a queue of simultaneous requests from the same or different users.

Finally, note that in the baseline agent approach and the others, we are making the assumption that an IRA processes one request to invoke one method on a single remote object. The intent of this assumption is to allow the IRA to be lean or thin, targeted to a very simple and easy to carry out task. There may be multiple IRAs active at the same time for a single UA, working on individual requests that will each invoke a single method on a single remote object. Having a single IRA travel to multiple locations isn't appropriate for the baseline agent approach, but may be relevant for the other approaches.

Object Security Agent The *object security agent (OSA)* can be conceptually regarded as the firewall that separates the remote object from the outside world. OSA embodies the security policy for the remote object, which in our case, is a role-based approach that dictates which role can access which method on the object [1, 2]. OSA receives a request to access a remote object via the mobile IRA, and based on the privileges of the role, will either deny or respond to the request. OSA is a gatekeeper to the remote object, but is important to again stress that the remote object may not be a single instance, but may be a collection of instances. OSA may manage a single instance (the "Steve" instance) or an entire collection (the "PersonDB" instance which contains instances for "Steve", "T.C", "Ying", "Mitch", etc.). Thus, a remote object is equivalent to a database in some situations.

The OSA component can be designed and implemented as either a set of security objects or a mobile agent. If OSA is a set of security objects, it is tightly bound to the class (type) of the remote object, and the supported security policy will be one of the object-oriented role-based techniques we have described in our earlier work [1]. Allocation of OSA in this situation will occur when the remote object is instantiated. If the OSA is an agent, it must be a separate entity from the remote object. The security policy that is supported in this situation can occur at the class (type) or object (instance) levels. In this case, there are three OSA allocation strategies. First, when the number of remote objects on a given computing node is small (or the number of objects that are typically accessed is small), then allocate a single OSA shared by all of the remote objects. Second, when the number of remote objects is moderate (or the number of objects that are typically accessed is moderate), then allocate an OSA for every remote object (instance). Third, when the number of remote objects is large, but the number of involved classes (types) is small, allocate an OSA for each class (type) of remote objects. There are clear tradeoffs in the aforementioned three strategies, particularly when one considers the frequency of access of remote objects by IRAs. For example, a single, shared OSA per server will quickly be overload by multiple IRA requests. Moreover, as activity increases and the numbers of mobile IRAs correspondingly increases, it is

likely that another component would be needed to oversee the interactions of IRAs with multiple OSAs.

2.2 HIERARCHICAL AGENT APPROACH

Figure 6.2 contains a representation of the architectural components for the hierarchical agent based approach which was designed to handle complex requests (nested transactions). While similar to Figure 11, the major difference is a hierarchical collection of mobile IRAs that are able to handle complex requests submitted by the client via UA. In the hierarchical agent approach, the logic in the IRA has been enhanced to support complex requests involving multiple remote objects on possibly different computing nodes. The hierarchical agent approach is appropriate when there are thin clients (e.g., a browser) and the processing of complex requests must be offloaded to either UA or IRA.

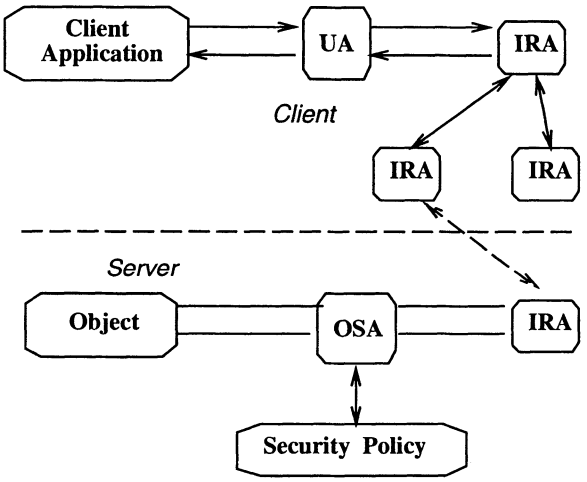

Figure 6.2 Architecture for hierarchical IRA approach.

To support complex requests, there are three different types of IRAs: *root-IRA, internal-IRA,* and *leaf-IRA*. A *root-IRA* is spawned by UA, and in turn can spawn a series of leaf-IRAs to handle complex requests. A *leaf-IRA* is similar in concept to the mobile IRA discussed in Section 2.1, and is intended to process the request for access to a remote object by moving across the network to another node. A *leaf-IRA* interacts with OSA to answer the user request, and returns its result to the root-IRA. Figure 6.2 depicts a situation where there is one root-IRA and two leaf-IRAs. The two leaf-IRAs in Figure 6.2 could move to the same or different computing nodes to accomplish their respective tasks. Upon their return, the root-IRA collects the results for forwarding to UA and eventually the user.

While Figure 6.2 only illustrates two levels of IRAs, in fact, there can be multiple levels, based on the complexity of the user request. If the root-IRA decomposes the user request into multiple complex sub-requests, an *internal-IRA* is spawned to handle complex sub-requests. Recursive spawning of IRAs will continue until the state of all leaf-IRAs is reached. As internal-IRAs and leaf-IRAs complete their actions, their responses will be collected by internal-IRAs, and eventually the root-IRA. In the aforementioned situation, the root-IRA is stationary; internal-IRAs can be either stationary or mobile.

¿From an allocation perspective, one approach would have a root-IRA allocated for each UA, whose lifetime is under the control of UA. Internal and leaf IRAs would be created by the root-IRA as needed to process user requests. Since only one root-IRA exists for each UA, the requests from the clients can be handled in a synchronized fashion. CA can submit complex requests to UA which are then passed to root- IRA for processing. The internal and leaf IRAs can process the sub-requests in a parallel fashion, by dispatching mobile agents (leaf-IRAs) to remote object locations.

We have adopted the strategy that each leaf-IRA will process a single user request (method invocation) on a single remote object. Our top-down strategy of root-IRAs spawning internal-IRAs spawning leaf-IRAs, is more suited to having each leaf-IRA visit a single remote object, and then require the internal-IRAs to collect and synthesize results in a bottom-up manner, eventually arriving at the root-IRA. A different strategy could have had each internal-IRA spawn a single leaf-IRA that would have a list of multiple user requests requiring access to potentially multiple remote objects. The single leaf-IRA could visit multiple servers in pursuit of all relevant remote objects. The tradeoffs are two-fold. First, the single leaf-IRA for an internal IRA may slow down processing, since the internal-IRA may need to get interim results from the single leaf-IRA before the same leaf-IRA is dispatched to other nodes. Second, the multiple leaf-IRAs may slow down processing at the client side with many agents that must be tracked and managed. There should be no impact on the server side in either situation, since the number of visits is the same; the identity (leaf-IRA vs. internal-IRA) of who is visiting is what has changed.

2.3 OSA MANAGER AGENT APPROACH

The *object-security manager agent approach* is presented in Figure 6.3, and expands the hierarchical agent approach by the addition of an OSA Manager. OSA Manager is responsible for arranging and overseeing the allocation of the OSA. Recall from Section 2.1 that we can allocate OSA in a number of ways: one per server when there are few remote objects, one per object for a moderate number of objects, or one per class, for a few classes that have many instances. An OSA Manager can dynamically pick and choose the best

allocation strategy for controlling access to remote objects during execution, in reaction to the state of the system, i.e., number of IRAs at client, number of remote objects being accessed, etc. The OSA agent approach extends the hierarchical approach by adding another layer of management at the server side to insure that the collections of remote objects are being managed in a fashion that is most suited to the execution environment of the running application.

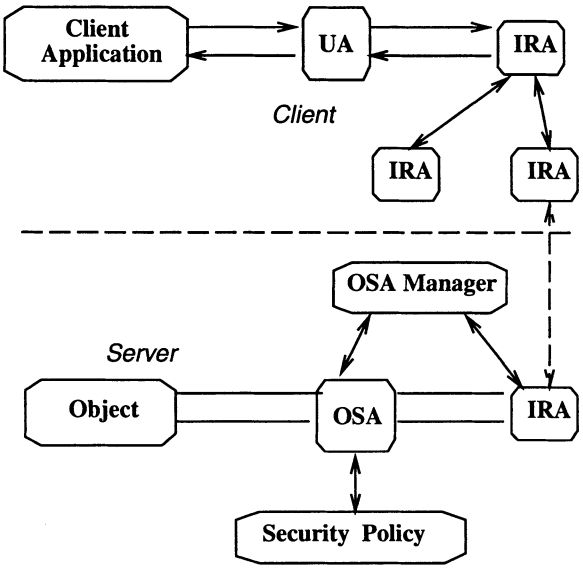

Figure 6.3 Architecture for OSA manager approach.

Processing by mobile IRAs must now be altered to communicate with OSA Manager to obtain information on the OSAs to be contacted for remote object access. That is, upon arriving at a server, an IRA asks OSA Manager where to find a specific remote object. OSA Manager returns the identity of the OSA that is controlling the remote object. Once the identity has been received, IRA contacts the correct OSA and proceeds to process the user request. The biggest advantage to an OSA Manager is to have tight and dynamic control over the OSAs that are resident on a server for processing user requests. Security policies are ever-changing, and the OSA Manager approach allows us to dynamically alter the allocation strategy for OSAs based on the system state.

2.4 SUMMARIZING APPROACHES: BASELINE VS. HIERARCHICAL VS. OSA MANAGER

The three approaches presented in Sections 2.1, 2.2, and 2.3 represent a progression based on a number of different dimensions. From the dimension of *client functionality*, the baseline agent approach is particularly well suited for legacy/COTS clients, where individual requests (method invocations) can be captured and sent to UA for processing. Since the functionality within the client in this situation is typically inaccessible (no source code), it makes sense to decompose the problem and send simple requests to UA which in turn create and dispatch IRAs as needed. The hierarchical and OSA Manager will work in this situation, but are better suited for thin clients (e.g., browsers), where significant application functionality is offloaded to UA and IRA. From the dimension of *UA allocation*, all three approaches are able to use either user-based or role-based allocation. The different allocation strategies are based more on client activity than on the agent behavior. From the dimension of *adaptability* of server to IRA frequency and load, the OSA Manager approach is superior. OSA Manager is intended to adapt the allocation of OSAs to remote objects based on the access patterns of the agents, ranging from one OSA/remote object to one OSA/class to one OSA/server. In the baseline and hierarchical approach, allocation of the OSAs would be fixed at system initialization. Finally, from the dimension of *complex request processing*, the hierarchical and OSA Manager approaches are equivalent in their ability to support nested transactions. All three approaches are controlling access to remote objects, which may range from small, individual instances (e.g., the Person "Steve") to collection instances (e.g., the PersonDB collection of Persons).

3. A JAVA AGLET IMPLEMENTATION

Stationary and mobile software agents in Java are supported by a number of different agent based systems, including Aglets [14], Odyssey [16], Concordia [15], and Voyager [17]. For our implementation we have chosen aglets, which was named by combining the terms of agent and applet. Unlike a Java applet, an aglet continues execution where it left off (upon reaching a new location). This is possible because an aglet is an executable object (containing both code and state data) that is able to move from computing node to computing node across a network. Like applets, aglet actions should be restricted to a Java sandbox. The sandbox model supports the running of untrusted code in a trusted environment so that if a hostile aglet is received, that aglet cannot damage the local machine. For applets, this security is enforced through three major components: the class loader, the bytecode verifier, and the security manager. Aglets would require the same level of security as applets. The

aglets would need to ask permission from the security manager before performing operations, thus allowing the security manager to know the identity of the aglet. This would be necessary for mobile IRAs that visit remote servers. We have prototyped two variants of the baseline agent approach, which differ in the OSA allocation strategies (see Section 2.1 again) and the number of involved computing nodes.

3.1 AGENT IMPLEMENTATION APPROACH

The agent implementation is shown in Figure 6.4. The main difference is a Translator that facilitates message passing communication between aglets. The Translator encodes the outgoing data from CA into a message (user request) that can be passed to UA, and retrieves data from the incoming message received by UA for the response to the request. In our prototype, a Translator is assigned to each CA/OSA and it is responsible for translating data for one user/remote object. In the implementation, the user's identity is included in every message. The Translator on the client side acquires the responsibility for authorization that was formerly performed by UA. The Translator on the server side includes the ability to invoke methods on remote objects. OSA passes IRA messages to its Translator, which will analyze the message, check the permissions to determine if the role can access the given method, and if so, invoke the method and create a reply message for OSA that is then passed to IRA. IRA moves back to the client, communicates via a message to UA, which in turn routes the response back to CA.

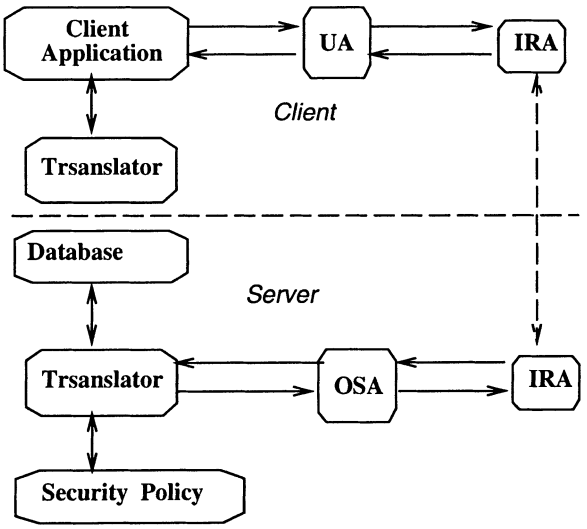

Figure 6.4 Architecture for agent implementation approach.

3.2 SAMPLE APPLICATION

The application involves typical university database access to courses, with Faculty and Student roles. Faculty has the permissions to add, delete, and find courses. Student can register, drop, and find courses. There are two remote objects: the Course database and the Person database. We experimented with two variants of the architecture discussed in Section 3.1 and shown in Figure 6.4. The first variant placed the Course and Person databases on the same server, and under the control of a single, shared OSA, thereby supporting the first OSA allocation strategy as given in Section 2.1. The second variant separated the two remote database objects, placing each on its own independent server, thereby requiring two separate dedicated OSA/Translator pairs.

3.3 ASSUMPTIONS: OSA DATABASE MANAGEMENT

In the first variant, a shared OSA is utilized to control the two databases. Based on our success, we went forward to adopt a more real-world approach in the second variant, with dedicated OSA/Translator pairs to control and manage databases on discrete servers. In our second solution, the management is accomplished by splitting the responsibility for data integrity across databases into the IRAs. IRA submits a request to either database via the OSA. Upon confirmation, from the OSA/Translator pair, a second request is forwarded to the other affected databases. Thus, a change in the Course database can be migrated to the Person database via the OSA/Translator.

3.4 EXPERIMENTATION IN GRADUATE COURSE

During the Spring 1999 semester at UConn, the aglet prototype for the baseline agent approach as discussed in Sections 3.1 to 3.3, was the subject of a course project. The students were presented lectures on both role-based security concepts and agent computing models, along with an examination of the concepts presented in this paper. With this as background, the students were given the aglet software consisting of 1500 lines of Java code. While the majority of the class was familiar with Java, no one in the class had programmed using agents and aglets. For the project, the students modified the agent-based implementation to includes a new Waiting List Agent (WLA), which is an application specific agent that would be responsible for monitoring the Course database for potential openings in full courses. The WLA would be spawned by CA as an option from the GUI. Once spawned, WLA would have a limited life time that is either user defined or tied to the last day of the add-drop period. WLA would be a mobile agent, free to move to the processing node

where the course database is resident. When an individual drops a course or seats are added to the course, WLA would automatically enroll the individual in the course and notify him/her by email. Surprisingly, all 24 students in the class successfully designed, implemented, and demonstrated this new agent, with 75% of the students able to send email when the course was added to a students schedule.

4. CONCLUDING REMARKS AND ONGOING EFFORTS

Our work in this paper has explored the utilization of mobile and stationary agents to define a set of architectural approaches in support of role-based security for dynamically accessing remote objects. We have presented a number of approaches in Section 2. that differ in their usage of agents and agent interactions, with a demonstration prototype reviewed in Section 3.. The three approaches were compared in Section 2.4 from various dimensions to clearly understand under which situations each approach is most useful. Overall, we believe that agent approaches to security are of increasing interest to organizations that are involved in web-based and distributed computing applications, for which the research community must provide apropos security solutions.

Our ongoing and future efforts in this area involve both continued prototyping and exploration of agent architectures. In Section 2., many different allocation strategies for the different components of our agent-security approaches were detailed. We are actively pursuing these other variants so that we can more precisely understand the tradeoffs in the different approaches. The architectural variants given in Section 2. are also being examined to consider other potential variants with different component structures. Finally, one glaring omission of our work is the realization of a security policy for a distributed setting. How is security for clients defined? For servers? For legacy/COTS? The ability to define, realize, and enforce such a policy is critical to insure that agents implementing security are working within a defined and refined context.

References

[1] Demurjian, S. and Ting, T.C. (1997). Towards a definitive paradigm for security in object-oriented systems and applications. *Journal of Computer Security*, **5(4)**.

[2] Demurjian, S., Ting, T.C. and Reisner, J. (1998). Software architectural alternatives for user role-based security policies. *Database Security, XI, Status and Prospects* (eds. T. Y. Lin and X. Qian), Chapman Hall.

[3] Genesereth, M. and Ketchpel, S. (1994). Software agents. *Communications of the ACM*, **37(7)**.

[4] Hale, J. *et al.*. (1996). A framework for high assurance security of distributed objects. *Proceedings of Tenth IFIP WG11.3 Working Conf. on Database Security*, Como, Italy.

[5] Hale, J. *et al.* (1998). Programmable security for object-oriented systems, *Proceedings of the Twelfth IFIP WG11.3 Working Conference on Database Security*, Chalkidiki, Greece.

[6] Karjoth, G., Lange, D. and Oshima, M. (1997). A security model for aglets", *IEEE Internet Computing*, **1(4)**.

[7] Keefe, T. *et al.* (1988). A multilevel security model for object-oriented systems. *Proceedings of the Eleventh National Computer Security Conference*.

[8] Landwehr, C. *et al.* (1984). A security model for military message systems. *ACM Transactions on Computer Systems*, **2(3)**.

[9] Lange, D. and Oshima, M. (1998). *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley.

[10] Lochovsky, F.H. and Woo, C.C. (1988). Role-based security in data base management systems. *Database Security: Status and Prospects* (ed. C. Landwehr), North-Holland.

[11] Sandhu, R. *et al.* (1996). Role-Based Access Control Models. *IEEE Computer*, **29(2)**.

[12] Smarkusky, D., Demurjian, S., Ting, T.C. and Bastarrica, C. (1999). Role-based security and Java. *Database Security, XII: Status and Prospects* (ed. S. Jajodia), Kluwer.

[13] Tari, Z. (1998) Designing security agents for DOK federated system. *Database Security, XI, Status and Prospects* (eds. T. Y. Lin and X. Qian), Chapman Hall.

[14] `www.trl.ibm.co.jp/aglets/`

[15] `www.metica.com/HSL/Projects/Concordia`

[16] `www.generalmagic.com/technology.technology.html`

[17] `www.objectspace.com/voyager`